Request Scheduling for Multiactive Objects

Justine Rochas

University of Nice Sophia Antipolis

Master IFI - Ubinet track

master thesis

Request Scheduling for Multiactive Objects

Student
Justine Rochas

Internship Supervisor
Ludovic Henrio

Academic Supervisor
Dino Lopez

Research Lab
INRIA-I3S-CNRS-UNS

Research Project-Team
OASIS

August 30, 2013

Abstract

The Active Object programming model aims to facilitate the writing of distributed applications. It provides asynchronous remote method calls and mechanisms to prevent data races. However, this model does not take advantage of multicore architectures as it is intrinsically mono-threaded. To overcome this problem, a recent extension of the active object model has been designed, which is called the Multiactive Object model. Multiactive objects enable local parallelism at a high level without giving up simplicity and safety provided by active objects: the programmer can declare which requests can be run in parallel through a customized specification language. However, in this new model, the programmer has not yet control on the scheduling policy applied to the requests executed by a multiactive object. In this work, we study application-level scheduling concepts that can be applied to multiactive objects. The goal is to allow the programmer to have a fine control on the scheduling policy through simple specifications. We first develop a priority mechanism that can be used to reorder awaiting requests in the queue of multiactive objects. Second, we provide a simple thread management mechanism to better allocate available threads to awaiting requests. Finally, we experiment the new features in a practical context, showing that the proposed mechanisms increase the efficiency of multiactive objects while keeping a low overhead. On the whole, we provide general specifications for application-level scheduling that are together fine-grained, user-friendly, and efficient.

Contents

Chapter 1

Introduction

1.1 Motivation

Writing distributed applications is tremendous. There are a lot of aspects in distributed programming that differ from local programming: naming, localization, timing, network reliance, etc. In order to ease distributed programming, specialized APIs, middlewares and distributed models are being developed. They attempt to hide remoteness the best they can by relying on generated code and fault tolerant protocols. Designing such protocols requires a deep understanding of language semantics. The object-oriented paradigm has been shown to be well adapted to abstract distribution, because (i) by definition, an object is something that can stand alone, which makes it easy to distribute, and because (ii) inheritance can be used to specialized a local method call into a remote method call. The RMI Java API [22] is a successful example of distribution abstraction; it provides classes that must be extended in order to enable remote object communication.

In the same objective, the Active Object model [18] is an example of programming model that is meant for distribution and that is suited to any object-oriented languages. In this model, objects are seen as independent entities that execute in their own thread. Active objects communicate with each other via asynchronous method calls, unlike lower level distribution mechanisms, like RMI (which supports only synchronous communications). In the particular implementation of active objects we study in this work, the same syntax is applied whether a method call is remote or local. In this case, the semantics of a method invocation is thus different whether it is remote or local, even if it is syntactically the same. Most of the distribution aspects are integrated when creating an active object: communication details are dealt separately from the business code. This completely hides distribution, which makes distributed code easier to write.

More recently, the same kind of abstraction emerged for concurrent programming. The goal is to give opportunity to the programmer to fully use the parallelism of computing units, without having to be an expert in concurrent programming. Again, static and dynamic analysis can be automatically made such that the parallelism is directly extracted from the code, or

simply declared by the programmer via a high level declaration language. For example, some C and C++ *pragmas* (preprocessor directives) [9] are devoted to parallelism and allow the programmer to indicate that some part of code should be parallelized at runtime according to several parameters. Here again, concurrency matters are isolated from business code, and producing parallel code is made easier.

In this objective, the OASIS research team has developed a new model built on top of the active object model, that gathers both distribution and concurrency abstractions. This new model is called the Multiactive Object model. It keeps the same convenience for distribution as active objects. In addition, this model provides facilities for the programmer to be able to define safe parallel executions within an active object. In this model, parallelism applies to requests (a multiactive object can execute several requests at the same time). Parallelism applies then to a higher level than *pragmas*, which usually apply to small parts of code. The concurrency model offered by multiactive objects is more convenient than existing low level concurrency mechanisms, such as locks and semaphores [10], because those tools are very permissive and error prone, whereas multiactive objects automatically block execution of requests that do not satisfy the specifications of the programmer. Thus, multiactive objects avoid data races that could be due to a lack of concurrency knowledge from the programmer.

The multiactive object model provides request-level parallelism thanks to the specifications written by the programmer. Currently, to execute requests, multiactive objects enforce an adapted FIFO policy for parallel safe execution. As multiactive objects rely on a limited number of threads to execute requests, it is possible that some requests wait in the queue even if they are ready to execute safely. As a consequence, it would be convenient to be able to reorder requests that are waiting in the ready queue according to their importance. But the programmer cannot assign yet priority of request execution in multiactive objects. The programmer cannot influence either the way the threads are allocated: if a ready-to-execute request is first in the queue, then it is allocated the next available thread, regardless of how many identical requests already execute. In brief, the scheduling policy applied in a multiactive object is not controllable from the programmer perspective. In this work, we attempt to empower the programmer with the ability to impact on the request scheduling of multiactive objects. The contributions of this work are threefold.

- First, we provide a new specification language to indicate priority of requests, and we show that it is easy to use and expressive.
- Second, we provide a mechanism to partition the available threads among the requests. Through this mechanism, we aim to avoid starvation of requests, assuming that this mechanism is correctly used.
- Third, we experiment those mechanisms in various ways, showing that they successfully increase the throughput of most important requests and that they do not introduce a

considerable overhead.

In summary, we provide efficient tools that allow the programmer to have more control on the scheduling policy of multiactive objects. As a consequence, those tools increase efficiency of multiactive objects at the application level.

1.2 Organization of the report

The report is organized as follows. Chapter 2 introduces the background, which is, on one hand, the active object model and its main existing implementations, and on the other hand, the multiactive object model and its current implementation. Then, the detailed objectives of the internship are presented considering the background. Finally, Chapter 2 exposes related works on high-level scheduling, either in the context of active objects or in the domain of meta programming.

Chapter 3 describes the contributions in detail. In particular, it introduces the two priority specifications that have been developed and states their main properties. It also exposes the thread allocation problem that arises from the introduction of priorities, and the solutions that have been given to this problem. Then, Chapter 3 exposes experimental evaluations showing that the priorities successfully speed up the processing of high priority requests. We also show that our main priority specification keeps a low overhead compared to the most efficient priority specification.

Finally, Chapter 4 concludes and gives an outline for future works.

Chapter 2

Background and Related Works

2.1 Active Objects

In this section, we first present in details the principles of active objects, and more precisely the particular implementation of active objects that is the basis of our work, which is the ASP [8] language and its implementation called ProActive [4]. Then, we review other existing active object implementations and compare them with our definition.

2.1.1 Principles

The active object model is a programming model that facilitates the writing of distributed applications. This model is not tight to a particular programming language. It is rather a design pattern for distribution that can be applied to any object-oriented language. In this model, certain objects can be marked as "active", so that their methods can be invoked from other objects located on remote machines, assuming that the machines are accessible through the network. In this model, the code related to the network calls is mainly generated and does not appear in the business code, which makes the distribution hidden from the programmer point of view. But active objects provide more than remote method call facilities, which is what differentiates them from lower level remote call mechanisms such as RPC (Remote Procedure Call) [7] or in an object-oriented context, RMI (Remote Method Invocation) [22]. Indeed, active objects have other properties that ensure both loose coupling of remote entities and safety of execution. More precisely, the three main characteristics of active objects are:

- Asynchronous remote method calls
- Absence of local concurrency of requests
- Absence of shared memory between active objects

Those characteristics differ from one implementation of active objects to another. We present below the particular implementation of active objects which is the basis of our work, that is called ASP [8].

The first characteristic of active objects is that they provide asynchronous remote calls: the remote calls are not blocking from the program that performs them. It means that the calling program might continue its execution before receiving the result of the call, if the result is not needed right away. Instead of the actual result, a promise of response is given to the caller. This promise of response is known as a *future* [12].

A future is an object that is a placeholder for the return value of the call. It is created transparently and behaves like the return value (i.e. it inherits from the return value). A future contains code to handle the particular situation in which the result is accessed after the call but before the reception of the result. This is how asynchronous calls are made possible. The synchronization occurs then implicitly when accessing the future. There are two kinds of accesses when reading a future. The first kind of access is called a non strict operation, and lets the future behave in place of the result. For example, a non strict operation is an affectation, or the fact to send the parameter of method call, i.e. when the value of the object is not truly needed, only its reference is needed. The second kind of access is called strict operation, and requires the future to block the execution until the result is received, if it is not received already. This is where the synchronization implicitly takes place if needed. For example, a strict operation is a field access or a method call on the result, i.e when the actual value of the result is needed. Figure 2.1 is an example of asynchronous call when using active objects.

```
1  // Suppose that actObj is an active object
2  Result res = actObj.produceResult();  // This triggers an asynchronous call
3
4  // Both following instructions do not block even if res is not received
5  // They are non strict operations
6  Result resSaver = res;
7  resUser.foo(res);
8
9  // This instruction blocks until the value of res is received
10 // It is a strict operation
11 res.bar();
```

Figure 2.1: Code illustrating strict and non strict operations on a future

At line 2, a remote call is transparently performed on **actObj**. The line 6 is executed just after line 2 (in fact, just after the request is received by **actObj**), without waiting for the **res** variable to be updated. Instead, a future object inheriting of the Result class is put in the **res** variable. Likewise, line 7 is executed without blocking, the future object is actually given as parameter of the method call. In all cases, the execution continues up to line 11 even if the value of **res** is not received up to this point. Line 11 is an implicit synchronization barrier because the actual value of **res** is needed to perform the **bar** method on it. The execution here blocks automatically until the value of **res** is received. In this case, it is the future itself that is in charge of blocking the execution; indeed, the future contains code to handle this case. This behavior is called *wait-by-necessity*, because the execution only blocks when truly

needed. Note that the behavior of the program is still predictable when using futures because it has a semantics close to a sequential execution, thanks to the wait-by-necessity property and thanks to the language design itself.

In the following we will talk more generally about *requests* instead of method calls, because this terminology better fits to the asynchronous characteristic.

The second characteristic of active objects is the absence of local concurrency. Indeed, the active object model ensures that requests that are sent to an active object are executed sequentially, even if several requests are received "at the same time"; at each moment, a single request is active. This is possible because an active object embeds a request queue (see Figure 2.2). Whenever a request is sent to the active object, this request is automatically

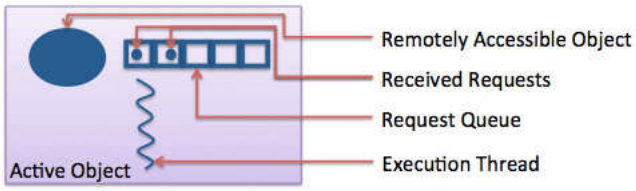


Figure 2.2: Anatomy of an active object

put in the queue. Then, the thread dedicated to the active object executes requests that are in the queue, one at a time. In the particular implementation we describe, when a request body starts executing, it must be finished before starting to execute another request body. There is absolutely no overlapping of request executions within an active object. This simple rule ensures that no data race can exist within an active object. A data race is a concurrent access to a shared resource that can lead to inconsistent state of the resource. For example, if two operations that assign a value to the same variable occur concurrently, the result is unpredictable since it depends on the interleaving of the operations. Such situations cannot arise in mono-threaded environments. This is why active objects enforce a strict sequential execution of requests, so that no data race can then exist.

The third characteristic is the absence of shared memory. Each active object has its own local memory and cannot access the local memory of another active object. In extension, a regular object can only be access by at most one active object. As a consequence, objects can only exchange information with objects of a non local memory via requests sent to active objects. This property makes object accesses easier to check and thus very safe.

In summary, when using active objects to write distributed applications, the programmer does not need (i) to write the code that performs the remote call and that waits for the reply (because this is handled transparently by the active object), (ii) neither to take care of concurrency problems that can arise when several requests are received at the same time.

Both mechanisms facilitate the writing of distributed code and ensure that a correct output is always produced.

From now we are going to focus on the second characteristic of active objects, that is absence of local concurrency, because our work applies to this particular characteristic. Then, we will see in Section 2.2.1 how this strict model is partially released to reintroduce local concurrency without loosing safety of execution. The goal is clearly to gain in efficiency.

2.1.2 An overview of active object languages

In this section, we review existing programming languages that implement the active object model. They have slight differences, firstly in the way the asynchronous calls are handled, and secondly in the way the absence of local concurrency is handled. Here we will mainly focus on the second point as it is the most relevant in the objective of designing scheduling policies.

Creol

In the Creol language [15], all objects are intrinsically active objects. It means that all objects have a queue where incoming requests are put and a dedicated thread to execute requests. As a consequence, all method calls are asynchronous: each method invocation is treated like a remote call. For this reason, Creol cannot rely on the sequential execution of requests of active objects, because this would be very likely to lead to a deadlock. Indeed, for example, if the completion of a request relies directly or indirectly on the completion of another request of the same object, then the execution of this request will never end.

So instead of sequential execution of requests, Creol proposes an instruction, called *await*, to explicitly release the execution thread. Any request body can call this instruction to let another request progress. The execution thread is then non deterministically allocated to a request that is waiting in the queue or to a request that yielded the execution thread before and that is now ready to resume. This kind of scheduling differs slightly from the active object model we presented. It is called *cooperative* (because requests can progress in an interleaved manner) *non preemptive* (because the execution thread is not yielded until there is an instruction await encountered) *scheduling*. So request executions can be interleaved but they never progress at the same time, since there still is only one thread to execute requests. In practice, there is no race condition but interleaving of requests can be complex: requests can be interrupted at release points, interleaved with other requests, and then resumed.

This design has two pitfalls. First, if the programmer does not place enough release points, this could result in deadlocks. Second, the programmer could place too many release points, resulting in a complex interleaving that could lead to an inconsistency. On the whole, safety of execution relies on the location of release points, that must be placed by the programmer. As a consequence, it is more difficult to program safely in this language.

JCoBox

JCoBox [21] is a Java extension that also enables asynchronous method calls. In this model, objects are organized into groups, called *CoBoxes*, such that each group is responsible to maintain its own coherent concurrency model. In this sense, JCoBox implements the active object model at the CoBox level instead of the object level. A CoBox can contain several objects, and objects in JCoBox can be of three kinds: active objects (accessible from another CoBox), regular objects (local to one CoBox) and immutable objects (objects shared from all active objects). The interface of a CoBox that is accessible remotely is the union of the interface of the active objects in this CoBox.

A CoBox has one request queue that contains all requests of its active objects. As in Creol (described above), requests are executed in a cooperative non preemptive manner, thanks to release points that the programmer is free to place wherever he wants. So in JCoBox, not only the requests of an active objects can be interleaved, but they can also be interleaved with requests of other active objects that lie in the same CoBox. However, contrarily to Creol that executes requests in a non deterministic manner, JCoBox enforces a FIFO scheduling policy in the waiting queue, where incoming requests and ready-to-resume requests are put together.

ABS

The Abstract Behavioral Specification (ABS) language [16] comes from the HATS [1] project and aims to provide tools for trustworthy distributed systems. ABS is a language of its own but can be translated into the Maude or Java languages for a better interoperability. ABS extends the Creol and JCoBox languages (described above) in many aspects. ABS provides groups of objects, as in JCoBox, that form coarse-grained remotely accessible entities. Such groups are called *cogs* in ABS. But contrarily to JCoBox, there is no objects shared among all cogs. Also, ABS provides cooperative non preemptive request execution of requests at the cog level: only one request is progressing at a time, but requests of a cog might have an interleaved execution.

However, as in Creol, there is no particular ordering of requests. Whenever the execution thread must be allocated, a request is chosen non deterministically among the waiting request set and and among the ready-to-resume request set. Thus, it is not ensured that the oldest request is executed, neither other deterministic behavior. In our opinion, this is be a drawback when some requests are important because it not known when they will be executed.

ASP

ASP (Asynchronous Sequential Processes) [8] is the active object language that follows the exact definition we gave of active objects in Section 2.1.1. Both regular and active objects cohabit. Each regular object is accessible via a single active object: there is no shared memory. Each active object has its own request queue and handles its own remote calls. There is

only one thread of execution and requests are executed one after the other. ASP enforces a strict sequential request execution, no interleaving at all. In addition, requests are selected deterministically for execution, by default according to a FIFO order.

In the following, we will focus on ASP active object language, as multiactive objects inherit from all of the features offered by ASP. ASP has a practical implementation in Java, called ProActive [4] that has recently been updated to implement multiactive objects. It is this particular implementation that has been modified and used in this work. In the following section, we detail both the multiactive object principles and its current implementation.

2.2 Multiactive Objects

2.2.1 Principles

The Multiactive Object model [14] is a multi-threaded extension of the active object model. It enables parallelism within an active object. This parallelism works at the request level: several requests can be executed at the same time. This is different from the cooperative non preemptive scheduling offered by other active object languages. Indeed, cooperative non preemptive scheduling interleaves requests, but there is never more than one request that progresses at a time. In the opposite, multiactive objects introduce a true parallelism. The main point of multiactive objects is to take advantage of multicore architectures. Indeed, active objects cannot take advantage of multicore architectures because they have only one single thread active a time (for safety reasons, as explained in Section 2.1.1). It means that a single processing unit is used over the whole life cycle of an active object. One solution to this problem could be to put several active objects on the same machine to load all the processing units. But this would imply performing remote calls instead of local ones to active objects that lie on the same machine, which is costly. This is a limitation that multiactive objects overcome. The multiactive object programming model aims at keeping the benefits of active objects in terms of ease of programming and safety of execution, while introducing parallelism in a simple and controlled way.

To keep concurrent execution of requests safe enough, multiactive objects rely on a meta language in order to specify which requests can be executed concurrently. The programmer can use this language to define requests that are *compatible* for concurrent execution. So here, safety of execution is partially handed over to the programmer. But specifying compatibilities is much easier than dealing with low level concurrency mechanisms, where each shared variable and critical section should be exhaustively protected by a lock.

In the current implementation of multiactive objects, the meta language that is used to specify compatibilities of requests has been developed using the Java annotation mechanism [2]. The Java annotation mechanism offers the possibility to define its own meta data structures and then to use them in the source code. Annotations can then be processed at compilation

time or at runtime. In the multiactive object case, annotations are processed at runtime to decide whether a request can be executed in parallel with others, which enables dynamic compatibilities.

As the work presented in Section 3 will use a similar mechanism to define scheduling controls, we are going to detail the annotations of multiactive objects that are meant to define request compatibilities. As a usecase, assume a distributed peer-to-peer system. Each peer in this system can be likened to a multiactive object that can handle several requests (lookup for a content, add a new content, handle a joining peer, route a request...). Figure 2.3 displays a typical definition of class Peer.

```
1 public class Peer {
2   public JoinResponse join(Peer other) { ... }
3   public void add(Key k, Serializable value) { ... }
4   public Serializable lookup(Key k) { ... }
5   public void monitor() { ... }
6 }
```

Figure 2.3: A typical Peer class definition in Java

Let us say that we want to parallelize the processing of requests that are received by a peer, using the multiactive object specification. Then three steps are required:

1 - @Group annotation The first step is to use **@DefineGroups** and **@Group** annotations on top of Peer class in order to define groups. A group represents a set of requests that are somehow semantically related. Defining groups aims at having a larger granularity than request granularity when specifiying compatibilities. Figure 2.4 shows an example of three group definitions. The first attribute of **@Group** specifies the name of the group. Here we define groups with names: **join**, **routing**, and **monitoring**. The second attribute of **@Group** specifies whether requests that belong to this group can be executed at the same time, i.e. whether the requests of the group are compatible with each other. For example, all requests of **monitoring** group can be executed together in parallel (**selfCompatible**=true), but a request of **join** group must be executed in exclusion with all requests of the same group (**selfCompatible=false**).

```
1 @DefineGroups({
2   @Group(name="join", selfCompatible=false),
3   @Group(name="routing", selfCompatible=true),
4   @Group(name="monitoring", selfCompatible=true)
5 })
```

Figure 2.4: An example of group definitions

2 - @MemberOf annotation The second step is to assign requests to groups using the **@MemberOf** annotation that applies on top of method definitions. Figure 2.5 shows how

methods of Peer are partitioned into previously defined groups. Here for example, the **join** request is the only one to be assigned to the **join** group. But both **add** and **lookup** requests are assigned to the **routing** group. Finally, the **monitor** request is assigned to the **monitoring** group.

```
1 @MemberOf("join")
2 public JoinResponse join(Peer other) { ... }
3 @MemberOf("routing")
4 public void add(Key k, Serializable value) { ... }
5 @MemberOf("routing")
6 public Serializable lookup(Key k) { ... }
7 @MemberOf("monitoring")
8 public void monitor() { ... }
```

Figure 2.5: An example of method assignments

3 - @Compatible annotation The last step is to specify which group is compatible with which other group (i.e. which requests are compatible with which others). **@DefineRules** and **@Compatible** annotations can be used on top of a class for this purpose. Figure 2.6 shows an example of compatibility definition. In this example, the first **@Compatible** line specifies that requests of the **join** group can be executed safely at the same time with requests of the **monitoring** group. The second line specifies the same for **routing** and **monitoring** groups. Any not specified combination leads to an incompatibility: for example, requests of the **join** group cannot be executed in parallel with requests of the **routing** group, because there is no **@Compatible** annotation that specifies this case.

```
1 @DefineRules({
2   @Compatible({"join", "monitoring"}),
3   @Compatible({"routing", "monitoring"}),
4 })
```

Figure 2.6: An example of compatibility definitions

The resulting Peer class is displayed on Figure 2.7. With a few lines of such annotations, the authors of [14] were able to speedup a particular scenario (involving a peer-to-peer system) with a factor 10.

2.2.2 Request service policy of multiactive objects

To efficiently execute requests according to defined compatibilities, multiactive objects enforce an adapted First In First Out policy with possibility to overtake. More precisely, a request that is waiting in the queue is executed if it is compatible with executing requests **and** with all previous requests in the queue. According to this definition, this scheduling policy is called First Compatible First Out. An example of execution is given on Figure 2.8, assuming the

```
@DefineGroups({
  @Group(name="join", selfCompatible=false),
  @Group(name="routing", selfCompatible=true),
  @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
  @Compatible({"join", "monitoring"),
  @Compatible({"routing", "monitoring"),
})
public class Peer {
  @MemberOf("join")
  public JoinResponse join(Peer other) { ... }
  @MemberOf("routing")
  public void add(Key k, Serializable value) { ... }
  @MemberOf("routing")
  public Serializable lookup(Key k) { ... }
  @MemberOf("monitoring")
  public void monitor() { ... }
}
```

Figure 2.7: Complete Peer class definition annotated for local parallelism

annotated Peer class. On this example, the first request in the queue cannot be executed because it is not compatible with executing requests. As the second request in the queue is compatible with both executing requests and the first request in the queue, then the second request can be executed right away in a new thread. Note that, with this rule, it is fair to execute the second request before the first one because the second request will not prevent the first one from executing (since they are compatible), so the first request in the queue will not be starved because the second one overtook it.

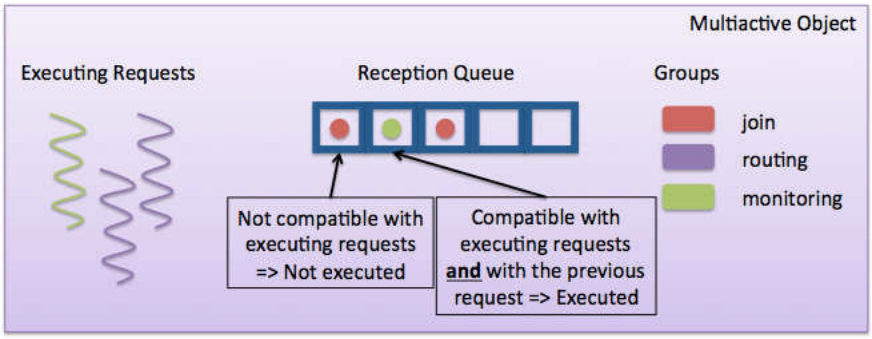


Figure 2.8: First Compatible First Out policy of multiactive objects

At any time, each ready-to-execute request should be assigned an execution thread. But to avoid killing the whole performance by creating too many threads on the fly, the number of threads that can be run at the same time can be limited. This limit can be set directly in the code. In this case, multiactive objects rely on a fixed thread pool to execute ready requests. As a consequence of this limit, even requests that are ready to execute (i.e. that are

compatible) might wait in a specific queue if all available threads are busy. Those requests that are ready to execute but that cannot be executed because of lack of threads form a new queue that we will call *ready queue*. Schematically, when a new request comes to an multiactive object, it is put in the *reception queue* of the multiactive object. Then, the compatibility filter is applied to select only the requests that are compatible, according to the policy describe earlier. Once filtered, those requests are put in the ready queue, waiting that some thread is freed to execute. In fact, the reception queue holds requests that cannot be served yet due to incompatibility, whereas the ready queue holds requests that are compatible (according to the above definition). Each request must go through both queues to be executed, since the scheduling of requests occurs at the head of the ready queue. This mechanism is pictured on Figure 2.9.

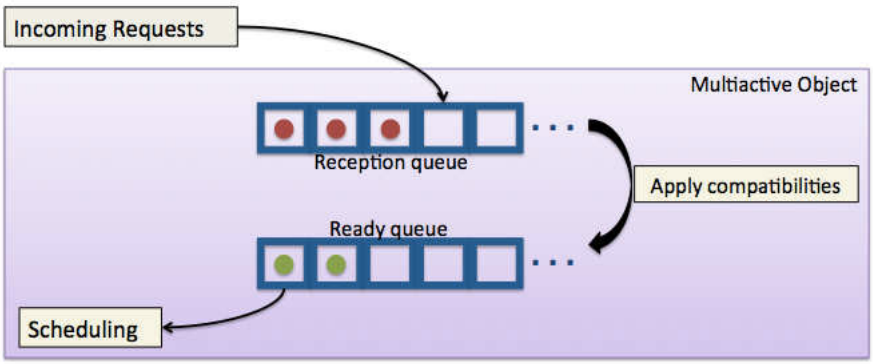


Figure 2.9: The different queues operating in a multiactive object

Considering the thread limitation, working on the ready queue and improve request scheduling is necessary to optimize request executions according to the programmer's needs. Indeed, introducing scheduling mechanisms at the reception queue level is useless since it is the compatibilities that first decide if a request is ready to be executed. For those reasons, all the objectives described below apply on the ready queue only.

2.2.3 Internship Objectives

The internship takes place in the context of multiactive objects. The main objective of the internship is to enhance the multiactive object programming model, by offering to the programmer more control on the scheduling of requests. To achieve this goal, we want to propose a priority mechanism applied to multiactive objects, in order to sort the requests in the ready queue according to their importance. We also want the programmer to be able to influence the scheduling of requests by setting limits on the number of threads that a type of request can occupy at the same time. The motivation here is to increase the efficiency of multiactive objects in the sense that the programmer will be able to favor processing of requests that are of most importance, while keeping a maximum parallelism and a safe execution. In particular, here are the main steps of the expected achievements:

1. Design, implement and analyze several priority specifications. The goal is to design a specification language for the programmer to be able to assign priorities to requests. This priority specification must be at the same time:

   - **User-friendly**
     The request priorities must be easy to define for the programmer and also easy to maintain (even with a high number of priorities to specify).
   - **Expressive**
     Any kind of priority dependency must be covered by the priority specification.
   - **Consistent**
     with the existing programming model, basically, using meta programming.
   - **Effective**
     The scheduling policy must take into account priorities as much as possible.
   - **Efficient**
     The priority specification must introduce a minimal additional cost to the execution of the scheduling policy.

2. Design and implement mechanisms to manage the allocation of threads. Propose tools for the programmer to be able to:

   - Set the size of the thread pool.
   - Reserve some threads for a particular type of requests.
     This tool can be used to specify that high priority requests have some threads devoted to them: this is a way to avoid the priority inversion problem.
   - Set a maximum limit on the number of threads that can be used by a particular type of requests at the same time.
     A good usage of this tool will ensure that low priority requests are not starved, i.e. that all requests in the queue are eventually executed.

3. Experiment the priority mechanism through test cases in order to:

   - Show that the priorities help to have a better throughput of most important requests: the priority mechanism is successfully applied.
   - Ensure that the priority mechanism does not have a too big overhead: the time to choose the next request to execute according to priorities must be reasonable.

Implementation of the proposed features will be done directly in the current implementation of multiactive objects, that is in the latest version of the ProActive Java middleware [4]. The OASIS research project-team is currently involved in a European project [3] that deals with automatic notification delivery through peer-to-peer networks using a Publish/Subscribe

mechanism. This system is currently implemented using multiactive objects from the ProActive implementation. Achievements of this internship objectives will contribute to improve the performance of this system. Experimentations will be carried out in this system, as it provides facilities to deploy multiactive objects.

The main challenge in this work is to find the most efficient specification that will improve expressiveness of multiactive objects, while keeping the model convenient for the programmer. In particular, finding the right balance between all the given objectives is the most difficult, because some objectives are orthogonal with each other.

2.3 Related works

In this section we review works that have been done in the domain of concurrency models or scheduling policies, in order to facilitate access to parallelism or to raise scheduling concerns at the application level.

2.3.1 JAC

JAC (Java with Annotated Concurrency) [13] is an extension of the Java language that aims at separating concurrency constructs from the application logic. JAC proposes a set of annotations that allow the programmer to specify whether the annotated entity must be guarded by a lock or if it can be run concurrently by several threads. This simplifies concurrency for non experimented programmers in this domain. More precisely, annotations are processed at precompile time (which limits runtime resolved decisions) and **synchronized** Java keywords are then introduced where needed before compilation. Most of the given annotations and facilities offered by JAC are very close to the ones offered by multiactive objects, especially to define method compatibilities. However, in multiactive objects, as the annotations for compatibilities are checked at runtime, compatibilities can be defined depending on the value of the request parameters, which cannot do JAC since its annotations are evaluated at precompile time.

In addition to compatibilities, JAC proposes a **@schedule** annotation, placed on top of a method, that allows the programmer to define a boolean method (linked to the annotated method) to decide whether the annotated method can be executed when invoked, in addition of compatibility rules. In this boolean method, the list of waiting requests is accessible, which makes possible to define fine-grained scheduling policies. However, this way to influence scheduling of request is fairly complex, as it requires the programmer to directly manipulate the requests in the queue. Unlike JAC, we want to offer very high level and intuitive scheduling mechanisms.

2.3.2 Scheduling in ABS

Although the ABS language (see Section 2.1.2) offers by default no deterministic policy to execute ready requests, ABS proposes to write user-defined schedulers that can be translated

into the Maude language (but not into Java), to override the non deterministic default policy. Thus, the programmer can define several customized schedulers and then specify one of them to be used in a particular cog. User-defined schedulers are methods written in the ABS language and they can be used through the **scheduler** ABS annotation that must mention the scheduler name. More precisely, a user-defined scheduler must select a request to execute among a list of requests waiting for a given object. As in ABS requests contain a lot of information (arrival date, waiting time, method call name, crucial method flag etc...), it is easy to design a customized scheduling policy based on any kind of criteria.

But there are two problems in this design according to our objectives. Firstly, a user-defined scheduler in ABS has not the last word about which request is actually executed: user-defined schedulers choose one request among waiting requests of a given object. But then, it is the cog that non deterministically decides which object is allowed to execute its elected request. So in the end, schedulers have only a partial impact on the scheduling. This violates our objective to have an effective user-defined scheduling. Secondly, user-defined schedulers in ABS still constrain the programmer to write code that manipulates the queue, which violates our user-friendly objective too.

2.3.3 Application-level scheduling in Creol

In this work [19], the authors extend the Creol language (Section 2.1.2) by introducing an application-level scheduling which relies on request priorities. Initially, in Creol, requests that are in the queue or that are waiting to resume are selected for execution in a non deterministic manner. This extension override this non deterministic behavior by assigning a priority to requests. Each request has either a user-defined priority or a default priority, materialized by an integer. The lower the integer is, the higher the priority is.

More precisely, the priorities of requests are assigned in two steps. First, a Creol active object interface must define a range of priority values. Second, when a method call is performed on an active object, the caller may assign a priority value in the range defined by the interface of the active object. Also, as the client side might always ask for the highest priority, the server side can also set a fixed priority value. So request priorities can be defined both on the server and client side. Then, to decide the final priority of a particular request, the active object applies a deterministic function that takes into account both the method call priority and the request definition priority.

Although this work is very interesting as it provides a high level scheduling hints, it is still complex to use because the programmer does not know right away the priority value of a request, because of server and client priority values, and because of the function that is applied on them. In our approach, we chose to assign priorities only on the server side, for the programmer to better visualize the priorities that will be actually applied, and to facilitate the use of priorities.

2.3.4 Positioning

Regarding related works about request scheduling, JAC gives the possibility to impact on the execution of requests by letting the programmer define particular methods on top of requests, to decide on their scheduling. Those methods are called just before trying to execute the request, to decide if it should actually be executed. ABS proposes a similar mechanism, known as user-defined schedulers, which are functions that manipulate the queue to select the request to execute. Although those solutions can provide a fine-grained selection of requests, because the programmer directly manipulates the queue, they are too low-level and they imply to write a significant part of the scheduling code. In our work, we prefer to provide mechanisms based on specification rather than programming, as they are more higher level and easier to use.

On the other hand, Application-level scheduling in Creol, which is the closest work to ours, provides request priority assignment, using a high level specification language. Priorities can be assigned both on the client and server side, for fairness reasons, and then a deterministic function combines all priorities to deduce the final one. This offers a very precise control of request execution. However, we consider that it is too complex because whenever the programmer assigns priority values, he has to be aware of all the values previously defined on both sides and to guess what the applied priority will be. He does not know right away the real priority value. In our work, we try to make the priority mechanism the easiest to use for the programmer. We want to keep both simplicity and expressiveness.

It is worth noticing that no related work considers at the same time distribution and parallelism. Indeed, JAC provides no facilities for distribution and ABS and Creol do not provide true parallelism of requests, just interleaved executions. On the other hand, multiactive objects provide distribution (via active objects) and true parallelism of requests (via the multi-threaded extension). Therefore, in addition of request execution order, we also give in this work scheduling tools to manage the threads in a programming model that is at the same time globally distributed and locally concurrent.

Chapter 3

Contributions

In this chapter, we present our contributions to the multiactive object model. We will look into priority of requests, thread management, and the validation of our main priority specification through experiments.

3.1 Design and implementation of scheduling controls

3.1.1 Priority specifications

Introduction

The multiactive object model enables multi-threading within an active object. While this model is convenient for completing multiple requests at the same time, it does not enable a particular scheduling other than First Compatible First Out, as explained in Section 2.2.1. But the programmer might appreciate some requests to be executed before some others if possible, that is, express a *priority* relationship between the requests. We can define priority relationship as the fact to reorder the requests in the ready queue such that requests which have a high priority can overtake requests which have a low priority. The main objective here is to reduce response time of most important requests.

Currently, in multiactive objects, requests lie in the reception queue in the order they have been received, that is a First In First Out ordering. Requests in the reception queue are then filtered such that compatible requests are pulled out to lie in a new queue that we call ready queue, as explained in Section 2.2.2. Requests are then executed according to the order of the ready queue. Currently, in the ready queue, a request is always placed after older requests and before more recent requests. Therefore, if in the ready queue a very important request is placed after a lot of unimportant requests, then the very important request has to wait the completion of all the unimportant requests before being processed. Prioritizing requests at the ready queue level aims at solving this problem by executing most important requests first, and this is what we introduce in this work. Criteria of importance can be of multiple sources. For example, in a storage system where updating the content is critical, we would like to be

able to postpone reading requests after modifying requests. Assuming that we are using an object as front end to store and retrieve content in this system, Figure 3.1 displays what could happen in this case with prioritized requests:

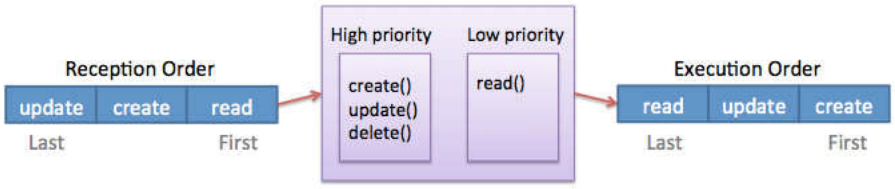


Figure 3.1: Priority principle: reorder request execution

In this example, there are four exposed requests: **create**, **read**, **update**, and **delete**. Those methods are partitioned into two groups (that we assume compatible with each other): the first one has a high priority and the second one has a low priority. Suppose that three requests are received for execution: first **read**, then **create**, and finally **update**. As we declared that the **read** request has the lowest priority, it is processed last, even if it was received first. It is this principle of priority that we want to address particularly in multiactive objects.

As we have seen in Section 2.2.1, a particular language has been designed using the JAVA annotation mechanism in order to allow the programmer to define compatibilities of requests. We have also used the Java annotation mechanism to offer to the programmer the possibility to define request priorities, to remain consistent with the existing framework. For that, in addition to the compatibility annotations, we have created new Java annotations for priorities. Those new annotations should also be located on top of a multiactive object class definition, and they should refer to the groups that the multiactive object declares. Priority annotations are then processed by the annotation handler that we have modified to adapt the scheduling of requests according to the defined priorities.

We have created a new simple declarative language that the programmer can use in order to easily assign priorities to requests. More precisely, we have chosen to assign priorities to groups instead of requests. Indeed, we keep the group granularity which is more general, still in our objective of being consistent with the existing programming model. As a consequence, all requests of the same group have the same priority. This does not restrict the possibilities of the mechanism: a group can still be split into several groups if needed. Note that, despite the fact that our particular implementation is described in the following, this priority model is not tight to a particular implementation of active objects. It can be implemented in any object-oriented programming language that supports meta programming.

In the following sections, we review two kinds of priority specification that we have developed. The first specification is based on integer assignment and the second one is based

on a dependency graph. We will analyze those specifications in order to converge towards a specification that meets most of the objectives we expect. Both specifications use customized annotations that we have developed in the current multiactive object implementation.

Integer-based priority specification

Integer assignment is a classical approach when dealing with priorities, mainly at the operating system level, but also at most of application-level scheduling systems. Integer assignment consists in associating an integer to a process/request/task, or more generally to entities in order to prioritize them. Thanks to this association, entities can then be ordered by importance, according to the meaning of the integers.

At the request level, integer assignment determines execution order of requests. As it is a classical approach to implement priorities, we have adapted this specification to our model. In our case, we want to assign a priority value to the requests of a given group. For that, we have developed two new annotations. The first one is called **@DefineIntegerBasedPriorities**. This global annotation is meant to contain several **@Priority** annotation entries. One **@Priority** entry corresponds to one level of priority. It accepts two attributes: first, a collection of group names previously defined, and second, a level of priority, materialized by an integer. This way, several groups can be given the same level of priority. Figure 3.2 is an example of how our annotations can be used to assign priorities to groups in the Java source, on top of a multiactive object class:

```
1 @DefineIntegerBasedPriorities({
2   @Priority(groupNames = {"G1"}, level = 5),
3   @Priority(groupNames = {"G2", "G3"}, level = 3),
4   @Priority(groupNames = {"G4"}, level = 2)
5 })
```

Figure 3.2: An example of priority annotations based on integer assignment

Here, four groups are given a priority, specifically **G1**, **G2**, **G3**, and **G4**. The highest priority requests are the ones of group **G1**, because it is assigned the highest value. Thus, in this example, requests of group **G1** will be the first to be allocated a thread if they are ready to be served (i.e. if they are compatible). Requests of groups **G2** and **G3** share the same level of priority and have a lower priority than group **G1**. Finally, requests of group **G4** have the lowest priority. As requests of groups **G2** and **G3** have the same priority level, they will be ordered together according to their arrival time. Thus, if requests of those groups are in the ready queue at the same time, they will be ordered according to their priority value: requests of group **G1** will be firsts, and requests of group **G4** will be lasts. Of course, we suppose here that all the groups mentioned in a **@Priority** annotations are previously declared in a **@Group** annotation.

The first advantage of this priority specification is that it is simple to process. Knowing

whether a request can be executed before another one consists in three elementary steps:

1. Look up group membership of the request
2. Look up the group priority value
3. Compare the value with the other request value

So the main operation to decide on the position of a request in the ready queue is an integer comparison. This operation is fast, since comparing two integers is fast. This is another advantage of this specification: it should have a good performance in practice as the priority processing should be here very fast. In our case, a good performance means a low overhead on the execution of the scheduling policy.

For requests of groups that do not have a priority value, an arbitrary default value is given. As well, requests that do not belong to any group have also a default priority value. In our implementation, the default value given is 0, and this value it is not customizable. Note that a request that does not belong to any group cannot have a user defined priority since priorities require group names. In the end, all requests have a priority value, even if they do not belong to any group nor have any priority.

As all requests have a priority value, we can always compare them to each other. Therefore, we can always order requests. If two requests have the same priority value, then we order them together following a FIFO policy. As a consequence of those two rules, we can always order requests in a strict total order. The insertion process of requests in the queue is the following. Assuming an incoming request R belonging to a group G, considering a ready queue composed of $R_1, R_2, ..., R_n$ belonging to groups $G_1, G_2, ..., G_n$; R is inserted in the ready queue just before the lowest R_i, $1 \leq i \leq n$, such that the priority value of G is greater than the priority value of G_i, if such a R_i exists. Else, it is inserted after R_n. More formally, we can define the insertion process as defined below:

Insertion process (integer-based) $group(R) = G \wedge Q = [R_1, ..., R_n] \wedge \forall i \in 1, ..., n$, $group(R_i) = G_i$

- $if\ priority(G_n) < priority(G)$,
  $let\ j = min(i|priority(G_i) < priority(G))$, $insert(R, Q) = [R_1, ..., R_{j-1}, R, R_j, ..., R_n]$
- $else\ insert(R, Q) = [R_1, ...R_n, R]$

Limitations of the integer-based priority specification

Always ordering the requests in a total order can be inconvenient. Even if the programmer does not assign a priority to all the groups, all requests are tight with each other with an order relation. But sometimes requests are not related at all, and there is no reason that one overtakes another in the ready queue. In this case we would like to keep the *unrelated*

```
@DefineIntegerBasedPriorities({
  @Priority(groupNames = {"G2"}, level = 2),
  @Priority(groupNames = {"G1"}, level = 1)
})
```

Figure 3.3: A priority definition involving two groups of different priority

relationship. As an example, consider the priority definition of Figure 3.3. Group **G2** has a higher priority than group **G1**. Aside, we want to specify that **G3** is not related with **G2** nor **G1**. However, with integer assignment, requests of group **G3** will always have either to wait after requests of at least one group or to overtake requests of at least one group, whatever the priority value assigned to it. This is inconvenient because this priority value will inevitably affect the throughput of requests of group **G2** and **G1**, whereas we want **G3** to be unrelated to them. Thus we cannot express the unrelated relationship using integer assignment. In this sense we can say that this priority specification language is not expressive enough to cover any usage.

Another drawback of this specification is that it is not user-friendly enough in our sense. Indeed, assigning a priority value to a new group implies to remember previously assigned values for all groups we want to overtake, simply to position the new group. Removing this indirection will allow the priority specification to be expressed more clearly. Instead, we expect a specification where the programmer simply says that he wants a group to have a higher priority than another group, without relying on additional knowledge, and without knowing how this fact is internally represented.

Clearly, this integer-based priority specification lacks of expressiveness and user-friendlyness. Those are the two reasons why we moved to another specification to define request priorities, that is more expressive and user-friendly. We describe this new specification in the next section.

Graph-based priority specification

In order to overcome the drawbacks of the integer-based priority specification, we developed a second model based on a dependency graph. The motivation of this specification is that it is able to express a relative relationship between groups instead of relying on absolute values. The idea is to implicitly define priority dependencies through the annotation itself, not through its attributes. In this approach, the annotations are processed in order to build a dependency graph, where nodes represent groups and directed edges represent a priority relation. This way, the internal representation of priorities is not exposed to the user, making the annotations higher level, and therefore, easier to use.

The new annotations we have developed are based on the following structure. First, a **@DefineGraphBasedPriorities** annotation is in charge of containing several dependency trees. Each dependency tree is defined in a **@PriorityOrder** annotation. A **@PriorityOrder**

annotation contains the group names, in an ordered manner using **@Set** annotations, such that a priority dependency exists from an upper line to a lower line. As a simple example, Figure 3.4 displays the same configuration as the previous one declared with the integer-based specification.

```
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2", "G3"}),
    @Set(groupNames = {"G4"})
  })
})
```

Figure 3.4: An example of priority annotations based on a dependency graph

Here, no value nor particular indication are given. It is only the order in which the groups are defined that is significant. Specifically, there is a priority dependency from **G1** to **G2** and **G3**, and from **G2** to **G4** and **G3** to **G4**.

Internally, when processed, this priority definition is stored as a dependency graph pictured on Figure 3.5. This example can be defined using only one **@PriorityOrder** annotation. But

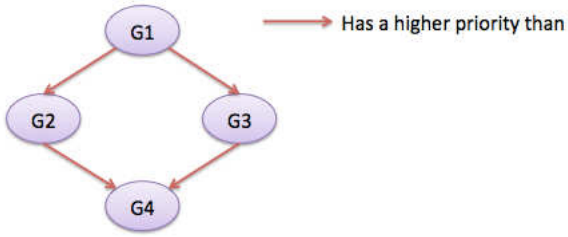


Figure 3.5: Dependency graph corresponding to the Figure 3.4 definition

the advantage of the graph-based priority specification is that we can define several branches of the graph separately without impacting the whole meaning. Figure 3.6 shows another example of the same graph definition.

In fact, one **@PriorityOrder** annotation corresponds to a linear representation of graph paths. With this specification, we can define the same priorities in different ways, according to the groups we want to focus on.

In the previous example, both the meaning and the impact of the priority definitions is exactly the same as the one defined with integer-based specification. And indeed, an integer-based priority specification can always be translated into a graph-based specification. However, a dependency graph cannot always be translated into a integer-based specification, because of the *unrelated* relation. Indeed, the unrelated relation is expressible with the graph specification but not with the integer specification, because of total order. Consider this particular dependency graph displayed on Figure 3.1.1.

```
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G4"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G3"}),
    @Set(groupNames = {"G4"})
  })
})
```

Figure 3.6: Another way to define the graph depicted on Figure 3.5

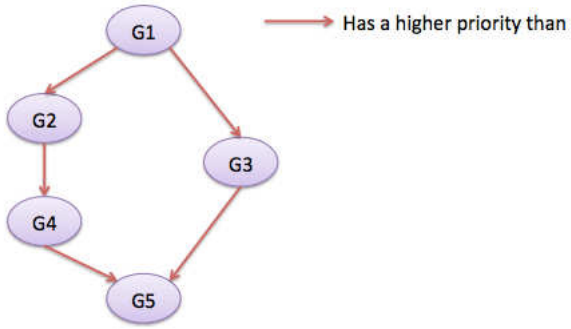


Figure 3.7: Dependency graph that cannot be translated using integer assignment

In this graph, there are two unrelated relations. In particular, **G3** is not related with **G2**, and **G3** is not related with **G4**. This is perfectly definable using our graph-based specification, by defining the two branches in separate **@PriorityOrder** annotations, as defined on Figure 3.8.

```
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G4"}),
    @Set(groupNames = {"G5"}),
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G3"}),
    @Set(groupNames = {"G5"})
  })
})
```

Figure 3.8: Graph-based definition of the graph depicted on Figure 3.1.1

However, this cannot be defined using integer assignment. Indeed, what value should be assigned to **G3** in order to not always overtake (or respectively, to not always be overtaken

by) **G2** nor **G4**? This idea is impossible to express with the integer-based specification. In the same idea, with the graph-based specification, there can exist **@PriorityOrder** annotations that lead to a disconnected graph. It is exactly what we want to be able to express, as to be able to express unrelated groups. To summarize, here are the points we have enhanced thanks to the graph-based specification:

- **More expressive**
  The set of priority relationships that can be expressed with the graph-based specification is larger than the set of priority relationships that can be expressed with the integer-based specification, because (i) the whole set of integer-based specifications is included in the set of graph-based specifications and (ii) there are graph representations that do not exist in the integer representation.

- **Easier to use**
  Branches of the graph can be defined separately, which makes the model more scalable. Even with a high number of user-defined priorities, more priorities can still be added easily without taking into account the existing set of priorities.

Graph properties

We have seen that the graph-based priority specification is more advantageous than the integer-based priority specification. We are going now to expose properties of our dependency graph that are essential to understand the scheduling policy.

Notations

Firstly, let us introduce the notations we will use in the following paragraphs.

- We start from a dependency graph defining a priority relation between groups, denoted with the operator $\longrightarrow$, whose operands are nodes of the graph.

- In the graph, reachability is denoted with the $\longrightarrow^+$ operator. The $\longrightarrow^+$ operator means that the second operand can be reached from the first one, i.e. that there exists a directed path from the first operand to the second one. The $\longrightarrow^+$ operator is the *transitive closure* of the $\longrightarrow$ operator.

- In our context, $G_1 \longrightarrow^+ G_2$ means that group G_1 has a higher priority than group G_2, since G_2 is reachable from G_1 in the dependency graph.

- A request is generally denoted R, a group denoted G and the ready queue denoted Q.

Absence of cycles

The dependency graph is inferred from the programmer's annotations. Thus, it is possible that the programmer, consciously or not, creates cycles in the dependency graph. For example,

```
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G1"})
  })
})
```

Figure 3.9: An example of priority definition that leads to a dependency cycle

Figure 3.9 is a syntactically correct priority declaration, but it creates a cycle that indefinitely goes from G_1 to G_2 and from G_2 to G_1.

Unchecked cycles in the dependency graph would lead to never ending search at some point, for example, infinite search of optimal position when inserting a new request. In order to prevent abnormal behavior related to graph cycles, we have enhanced our initial graph construction process. At each new annotation line encountered, we check if the new dependency will lead to a cycle in the graph. If so, the dependency is not created. If the node currently considered did not exist already, then it is not created either. If it existed already, following annotation lines are still taken into account according to the same rules. In case of a detected cycle in the annotations, an error message is displayed showing the incriminated dependencies. Thus, by construction, we ensure that there is no cycle in the dependency graph we build. In other words, by construction, our dependency graph is a Directed Acyclic Graph (DAG) [11].

Unicity

In our dependency graph, each group belongs to a single node. We ensure this fact by construction too: when a group name is encountered in a priority annotation, the algorithm first searches a node whose group has the same name. If such a node exists already in the graph, then no additional node is created. Only the references that point towards higher priority nodes and lower priority nodes are updated. When a new node is created because it does not exist already, the new node is stored such that it can be retrieved (by exploring the graph) if it is referenced later.

Overtakability

In our scheduling objective, the main information we want to extract from the graph is whether a request can overtake another one. More generally, as we operate on groups of requests, we are interested in knowing whether a group has a higher priority than another one. This information lies in the dependency graph.

- A request of group G_1 can overtake a request of group G_2 if and only if $G_1 \longrightarrow^+ G_2$, i.e. if it exists a directed path from G_1 to G_2.

- A request of group G_1 has no priority relation with a request of group G_2 if ($\neg$ $G_1 \longrightarrow^+ G_2 \ \wedge \ \neg \ G_2 \longrightarrow^+ G_1$) $\vee$ $G_i = G_j$, i.e. if there is no directed path from G_1 to G_2 and no directed path from G_2 to G_1, or if the same groups are considered. In this case we say $G_1 \ // \ G_2$.

Considering our insertion policy described below, what is important from those definitions is that G_1 cannot overtake G_2 if either $G_1 \ // \ G_2$ or $G_2 \longrightarrow^+ G_1$.

Insertion

In practice, when a new request is received and must be inserted in the ready queue, overtakability properties are used to determine the position of the new requests according to the priorities. In particular, an incoming request of group G must be inserted in the ready queue just before the first request whose group can be overtaken by the group of the inserted request. Precisely, considering an incoming request R belonging to group G, and a ready queue made of $R_1, R_2, ..., R_n$ belonging to groups $G_1, G_2, ..., G_n$, then R is inserted just before the smallest R_i, $1 \leq i \leq n$, such that $G \longrightarrow^+ G_i$, or at the end of the queue if $\forall$j, $1 \leq j \leq n$, $G_j \longrightarrow^+ G \ \vee \ G \ // \ G_j$. More formally, we can define the insertion process as defined below:

Insertion process (graph-based) $group(R) = G \ \wedge \ Q = [R_1, ..., R_n] \ \wedge \ \forall i \in 1, ..., n$, $group(R_i) = G_i$

1. $if \ \forall \ i \ G_i \longrightarrow^+ G \ \vee \ G \ // \ G_i$, $insert(R, Q) = [R_1, ...R_n, R]$
2. $else \ let \ j = min(i | G \longrightarrow^+ G_i)$, $insert(R, Q) = [R_1, ..., R_{j-1}, R, R_j, ..., R_n]$

Theorem 1. *The ready queue is always ordered such that if $i < j$, then $G_i \longrightarrow^+ G_j$ or $G_i \ // \ G_j$.*

Proof. First notice that for all groups G and G', either $G \longrightarrow^+ G'$, or $G' \longrightarrow^+ G$, or $G \ // \ G'$. Second we do the proof by recurrence on the states of Q along time, let us prove that the queue is always "well-ordered", i.e., if $i < j$, then $G_i \longrightarrow^+ G_j$ or $G_i \ // \ G_j$ (but not $G_j \longrightarrow^+ G_i$).

1. The queue is initially empty ($length(Q) = 0$), so the empty queue is trivially well-ordered.
2. When a new element is inserted, suppose $length(Q) = n, n \geq 0$ and Q is well-ordered. Suppose that we have an incoming request of group G to insert in $Q = [R_1, ..., R_n]$ with $R_1, ..., R_n$ respectively of groups $G_1, ..., G_n$. Let us look at the two cases of the insertion process.
   (a) Suppose that we did not find $j \leq n$ such that $j = min(i | G \longrightarrow^+ G_i)$. Then, by definition, $G_i \longrightarrow^+ G \ \vee \ G \ // \ G_i$, so the new queue is well-ordered. Indeed if $i < j \leq n$, then $G_i \longrightarrow^+ G_j$ or $G_i \ // \ G_j$ by recurrence hypothesis, and else $i < j = n + 1$ and $G_i \longrightarrow^+ G \ \vee \ G \ // \ G_i$.

(b) Suppose now that we found $k \leq n$ such that $k = min(i|G \longrightarrow^+ G_i)$. The new queue is $[R'_1, .., R'_{n+1}] = [R_1, .., R_{k-1}, R, R_k, ..., R_n]$. Let us prove that the new queue is well-ordered by considering two requests R'_i and R'_j in this queue. If $j \neq k$ and $i \neq k$ then the recurrence hypothesis is sufficient to conclude. Else we have two cases to consider: either $i = k < j$ or $i < j = k$. Let us prove by contradiction that the queue is still well-ordered after $insert(R, Q)$ in both cases.

i. $(i < k)$ If the queue was not well-ordered after $insert(R, Q)$, we would have $G \longrightarrow^+ G_i$ (see the first note of this proof). But this would be contradictory with the definition of k, which stipulates that k is the minimum i that satisfies $G \longrightarrow^+ G_i$.

ii. $(k < j)$ If the queue was not well-ordered after $insert(R, Q)$, we would have $G_{j-1} \longrightarrow^+ G$ (the element number j of the new queue corresponds to $R'_j = R_{j-1}$). By definition of k, we have also $G \longrightarrow^+ G_k$, so we have $G_{j-1} \longrightarrow^+ G \longrightarrow^+ G_k$. By transitivity, we have then also $G_{j-1} \longrightarrow^+ G_k$ which is contradictory with the fact that $k \leq j - 1$. Indeed, if $j - 1 = k$ then $G_{j-1} = G_k$, else by recurrence hypothesis (and by the first remark) $\neg G_{j-1} \longrightarrow^+ G_k$. This concludes about the contradiction in the second case.

3. Finally, it is trivial to check that the queue is still well-ordered when the head of the queue is dequeued.

□

To conclude, the graph-based priority specification is very convenient to define relative priorities between group of requests of multiactive objects. The programmer only defines dependencies between groups, that are then automatically taken into account as priorities when scheduling requests. This specification internally uses a dependency graph that is not visible from the user. This allows our graph to have strong properties. The only drawback of this specification is that it uses a quite complex structure (a graph) that (1) has to be kept in memory the entire execution time, and (2) might be entirely explored each time a request is inserted in the ready queue. The second point impacts on the scheduling time of requests since the queue must be locked during the whole insertion process to prevent data races. This can be a problem considering our efficiency objective, which stipulate that our priority specification should have a low overhead on the scheduling time. On the other hand, the integer-based specification seems to be well adapted for efficient processing. This is why, despite the convenience of the graph-based specification, we need to validate this specification through experiments that compare it with the integer-based specification. Such experiments will be presented in Section 3.2.

Note that from Java version 8 (that is not yet released - planned on early 2014), we will be able to get rid of the container annotations. In our implementation of priority specifications,

container annotations are named with the **@Define** prefix, as in **@DefineIntegerBasedPriorities** and **@DefineGraphBasedPriorities**. Indeed, Java 8 will introduce the possibilty to use an annotation more than once on top of a given entity, which is not allowed for the moment. This is why we had to use container annotations to encapsulate several declarations, which makes the priority specification a little bit heavier. The contrast between both versions is emphasized in Appendix B.

3.1.2 Thread management

Problematic

We increased the control on the scheduling of requests in multiactive objects by offering to the programmer a powerful priority specification. But to offer further control on the scheduling we also provide a mechanism to manage threads that are available to execute the requests. Indeed, the way the threads are allocated to execute the requests is not currently customizable in multiactive objects. As soon as a request is selected for execution, it is allocated the first available thread. In particular, no further checking is made about requests already executing, nor about waiting requests. The problematic is the following: considering that we have a limited number of threads and that we have introduced priorities to execute requests, then abnormal situations can arise. Especially, low priority requests can suffer of *starvation*. Starvation is a situation where access to a resource is perpetually delayed such that it will never be accessed. In our context, the resources are threads, and the entities that fight to access the threads are requests. For example, a starvation situation can occur in our case when a continuous stream of high priority requests comes, preventing low priority requests from being executed. This part of the work aims at being able to partition the threads among the different groups (i.e. to the different priority levels) according to the programmer's wish.

Starvation of low priority requests

To prevent starvation of low priority requests at the programming level, we have enhanced the existing group definition. The goal is to allow the programmer to set a limit on the number of threads that requests of a particular group can use at a time. Optionally, the **@Group** annotation can now declare an attribute that determines the maximum number of threads that requests of the group can occupy at the same time. Figure 3.10 displays an example of the new group annotation.

```
1 @DefineGroups({
2   @Group(name="G1", threadLimit=5, selfCompatible="true"),
3   @Group(name="G2", threadLimit=3, selfCompatible="true")
4 })
```

Figure 3.10: Enhanced group annotation taking into account thread limitation per group

In this example, the first **@Group** annotation specifies that requests of group **G1** cannot occupy more than 5 threads at the same time. Similarly, the second **@Group** annotation specifies that no more than 3 threads should be occupied by requests of group **G2** at the same time. This is a simple way to ensure that requests of a same group do not capture all the available threads, assuming that the programmer correctly uses the **threadLimit** attribute.

Starvation of high priority requests

The second starvation problem is known as *priority inversion* [17]. Here is its brief explanation: if a request with high priority is received, but all the available threads are already allocated, then this highest priority request has to wait whereas there might be lower priority requests that are currently executed. This situation, where low priority requests block execution of high priority requests, is unintended and corresponds to an inversion of priority. In order to handle this problem, we give the possibility to the programmer to reserve some threads (which is different from limiting the thread utilization). Such reserved threads are only dedicated to a given group. Reserved threads apply in addition to the maximum number of threads that the group can occupy. To support this feature, the **@Group** annotation has been once again enlarged. An optional attribute can be used to specify the minimum number of threads that should be kept free to execute requests of a given group, as shown in Figure 3.11:

```
1 @DefineGroups({
2   @Group(name="G1", threadLimit=5, reservedThreads=2, selfCompatible="true"),
3   @Group(name="G2", threadLimit=3, reservedThreads=1, selfCompatible="true")
4 })
```

Figure 3.11: Enhanced group annotation taking into account reserved threads per group

This specification ensures that always 2 threads are devoted to execute requests of group **G1**, and that 1 thread is devoted to execute requests of group **G2**. We can use this **reservedThreads** attribute to ensure that high priority requests have always some threads to execute them, thus preventing the priority inversion problem. The reserved threads must be included in the thread limit, which means that the number of reserved threads must be lower than the maximum number of threads that can be occupied at a time.

In fact, the **reservedThreads** attribute corresponds to the lower bound on the number of threads used at the same time by requests of a given group, provided that some requests of this group are ready to be served. On the other hand, the **threadLimit** attribute corresponds to the upper bound on the number of threads used by a group at the same time. Those mechanisms help to balance the distribution of the threads among groups of requests.

Remaining concerns

We have introduced mechanisms in multiactive objects that offer to the programmer the possibility to control thread allocation, in a fine-grained manner. As those mechanisms are purely declarative, they can be easily used. However, empowering the programmer with those tools is risky: a wrong partitioning could lead to starvation of requests. To prevent this problem, we offer a default behavior that overrides the programmer's specification if needed:

- If the number of reserved threads is higher than the maximum number of threads used at the same time, then the limiting value is assigned to both attributes.
- If the total sum of reserved threads for all groups is higher than the size of the thread pool, then we increase the size of the thread pool consequently.

A second drawback of the thread management mechanisms that we have developed is that, although practical and required to prevent starvation, they weaken the priority specification. Indeed, even the highest priority request cannot execute if the maximum number of threads authorized for its group is already reached. Figure 3.12 displays an example of such situation.

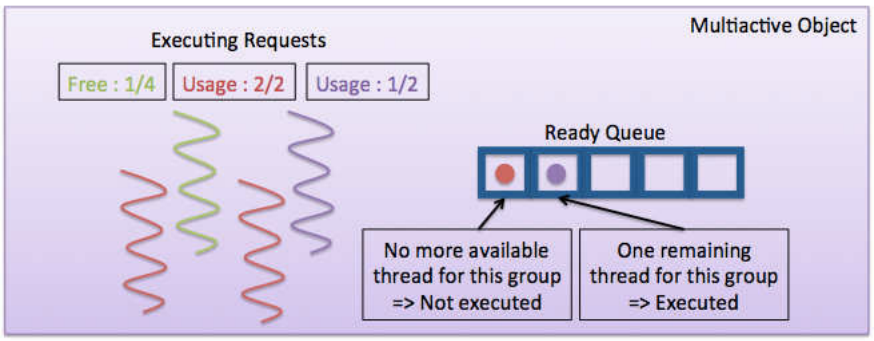


Figure 3.12: An example of request scheduling considering thread limits

Nevertheless, considering compatibilities, priorities and thread management, the scheduling of requests within a multiactive object is still deterministic, because the rules we apply are deterministic. In the end, priorities can be considered as hints of execution more than strong requirements, but this is needed to balance the execution. No policy can both ensure fairness of requests execution and strong priorities at the same time, because those notions are orthogonal. We attempted to converge towards a scheduling model that is well balanced between the programmer's wishes and a correct execution.

3.1.3 Software Architecture

In this section, we expose the integration of our scheduling mechanisms in the the global architecture of multiactive objects, as well as we present the main internal structures used to implement them.

A multiactive object intrinsincally embeds an object of type MultiactiveService. The MultiactiveService has two purposes. Firstly, it relies on an AnnotationProcessor that processes annotations to initialize all the structures required by a multiactive object (structures that store compatibilities, priorities and keep tracks of thread utilization). Secondly, the MultiactiveService has a RequestExecutor, that schedules requests according to compatibilities, priorities and available threads. All manager classes rely on the structures previously established when reading annotations. Figure 3.13 displays an UML diagram describing the class composition of a multiactive object, as well as the most important interfaces.

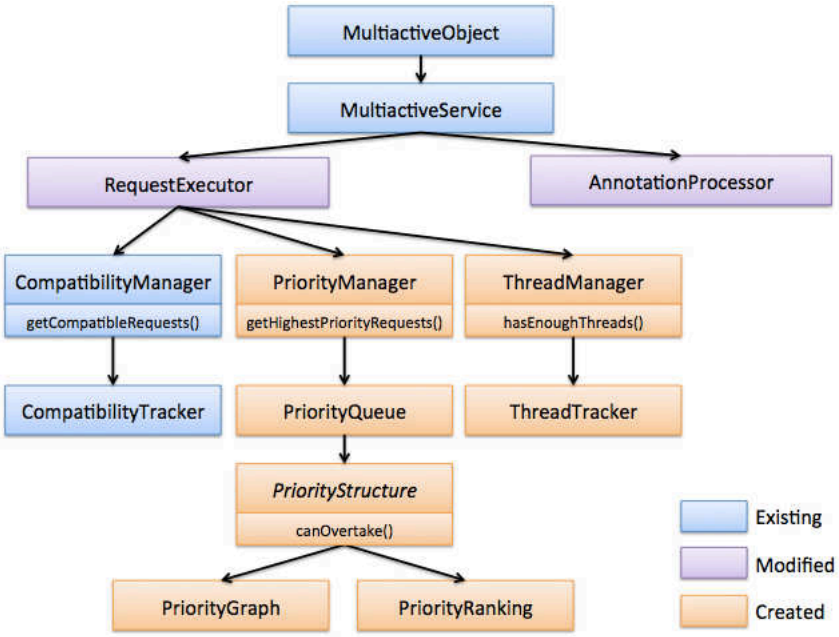


Figure 3.13: Class diagram of multiactive objects with scheduling controls

The RequestExecutor has only access to the interface of the CompatibilityManager, the PriorityManager and the ThreadManager. Each entity manages its own state internally. More precisely, to execute requests, the RequestExecutor first filters incoming requests with the CompatibilityManager. Among compatible requests (request that are ready to be executed safely), a second filter is applied using the PriorityManager, that sends back a list of the highest priority requests. Then, the final decision regarding the execution of a request depends on the ThreadManager, that filters highest priority requests according to the number of threads that are available for them. After applying those three filters sequentially, remaining requests are the ones that should be actually executed. Indeed, each request must satisfy the three steps of compatibilty, priority, and thread availability before being executed.

The priority mechanism is the most complex. It relies on a customized PriorityQueue (unfortunately, existing implementations could not be used because of specific needs). The PriorityQueue adheres to the producer/consumer design pattern [5]. On one hand, a dedicated thread registers compatible requests to the queue. On the other hand, another dedicated thread

polls the queue to retrieve highest priority requests. This synchronization is implemented using Java's concurrency mechanisms. When a request is registered to the PriorityQueue, the PriorityStructure interface is questioned to know where the request should be inserted. The concrete PriorityStructure used for that can be either our dependency graph (PriorityGraph), or our integer-based priority specification (PriorityRanking). A concrete PriorityStructure should implement the **canOvertake(request1, request2)** method that says whether a request can be executed before another one. In the PriorityGraph case, this method goes through the graph to retrieve this information. In the PriorityRanking case, the priority values of the requests are simply retrieved from a tree-based map, which ensures a good efficiency.

This design can easily be adapted to other implementations of active objects, that basically support annotations and their processing using reflexivity.

3.2 Experimental evaluation

In this section, we evaluate the performance of our priority mechanisms through experiments. We show first that our main priority specification is effective: priorities contribute to speed up the processing of high priority requests. Secondly, we show that the graph-based specification is efficient: the cost to reorder the ready queue according to priorities is not too high, and we even manage to reduce significantly this cost by improving the internal representation of the graph.

3.2.1 Environment

All experiments are run on a single machine because we want to evaluate parallelism of requests, which occurs within a single multiactive object. Thus, even though the context of multiactive objects is distributed, we only need to run experiments on one machine, because our scheduling mechanisms only apply on request parallelism. It is worth noticing that our scheduling mechanisms contribute to the efficiency of local parallelism, not to the efficiency of distributed execution.

The machine used to run our experiments has four 4-core Intel Core Q6600 processors and 8GB of memory. Experiments are run using a Java 7 virtual machine. We developed our experiments using the EventCloud platform [20] because it provides an API to easily deploy multiactive objects. For that, the EventCloud platform relies on the Proactive middleware [4] that implements multiactive objects. We have modified the Proactive middleware to introduce our scheduling mechanisms, and we have then used this customized version as a basis of EventCloud.

In order to run our experiments, we develop a dedicated multiactive object class. We define its methods and we annotate the class according to the experiment we want to run: we create groups, we assign methods to groups and we assign priorities to groups. To experiment in

neutral conditions, all method bodies contain just a few log instructions and a return body. Such small methods allows us to compare the smallest execution time with the overhead of priorities. In all experiments, all groups are declared compatible, otherwise, they would never be in the ready queue at the same time, and we could never apply priorities. The general process of each experiment is the following. We first create and deploy a single multiactive object. Then, we send requests to it, and we record relevant metrics according to the current experiment, mainly using logs. Finally, we process the logs and get our results. In the following, when the priority specification that is used is not clearly mentioned, it means that we are using our graph-based specification to assign priorities.

Following experiments test our priority specification in a worst-case scenario. Firstly, all requests that we send to the multiactive object have the smallest possible body. Indeed, all methods are made of at most two instructions: a logging instruction and a return instruction. In practice, requests sent to a multiactive object should be long enough to be balanced with the communication time. As a consequence, in practice, the performance of multiactive objects should be higher than in our experimental situations. Secondly, we intentionally block the execution until all requests we want to experiment are received by the multiactive object and put in its queue. This way, we end up having a big ready queue, which again makes the scenario the worst possible. In a real application, a multiactive object that receives a hundred requests in a short period of time is already an unusual case. To block the execution until all requests are received, before each experiment, we send to the multiactive object one big (long) request per available thread. Those requests occupy all the available thread long enough to ensures that the requests we actually evaluate are all inserted in the queue before we start executing them. In brief, our experiments test the priority specifications at their limits, and the performance can only be better in real cases.

3.2.2 Speed up of high priority requests

In this experiment, we want to show the effectiveness of our priority specification. We define only two groups: A and B. Each group has only one assigned method. We send in total 1000 requests to the multiactive object that we have previously defined. More precisely, we send alternatively a request of group A and a request of group B, so 500 requests are sent for each group in total. We set a thread pool that contains four threads. We set no thread limitation for group A, neither for group B.

We run two executions of this scenario. In the first execution, we do not declare any priority relation between group A and group B. In the second execution, we declare that group A has a higher priority than group B. For both executions, we evaluate the execution time dynamic of requests of group A and requests of group B. Here, we define the execution time as the sum of:

- The time to send the request to the multiactive object (registration time)

- The time to insert the request in the ready queue (insertion time)
- The time spent in the ready queue (waiting time)
- The time to serve the request (service time)

To perform this evaluation, we average execution times of requests per batch of 50 consecutive requests of the same group. Figure 3.14 depicts the results with prioritized and not prioritized executions. When not using priorities, the execution time is the same for both

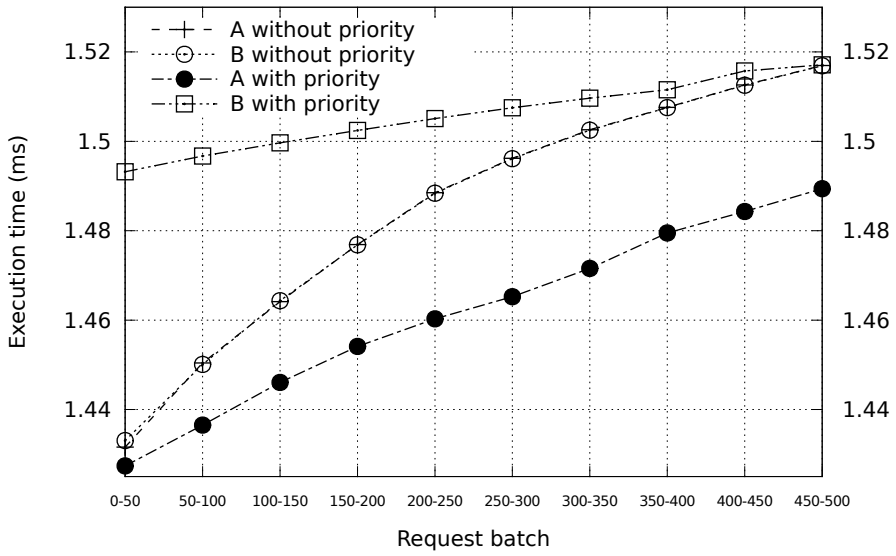


Figure 3.14: Execution time of A and B requests with and without priority

groups. It increases equally as the ready queue increases, because requests are executed in the same order as they are received. Consequently, the waiting time of requests of both groups increases in the same way. Also, in this case the insertion time is the same for both groups because no priority applies.

On the other hand, when using priorities, the execution time of requests of group A grows less quickly than the execution time of requests of group B, as expected. This is due to two facts. First, requests of group A wait less than requests of group B, because they are positioned before in the ready queue. Second, the time to insert requests of group A in the ready queue is smaller than the time to insert requests of group B, because in the first case, only half of the ready queue is examined whereas the entire ready queue is examined when inserting a request of group B. The execution time of requests of group A when using priorities is always smaller than other execution times. This shows that our priority specification is effective: it successfully speeds up the execution of high priority requests.

3.2.3 Overhead of the different priority specifications

In this section, we perform two experiments that evaluate the *overhead* of priorities. What we call overhead here is the time needed to insert an incoming request in the ready queue, according to its priority. When no priority mechanism exists, such overhead is quasi null: the request is just put at the end of the queue, which is a constant operation. However, with the priority mechanisms we have developed, we have seen that we potentially have to look through the whole ready queue to find the right position of an incoming request. As a consequence, this operation is dependent on the number of requests that are already in the ready queue. In addition, the structures used to store priorities of requests is also a factor of complexity of the insertion process. In our case, we have two priority specifications: the integer-based specification and the graph-based specification. They rely on two different structures: a linear structure and a graph structure. We evaluate both of them to see if the overhead of our graph-based specification is not too high compared to an efficient specification (the integer-based specification). In the two following experiments, we set a thread pool that contains only one thread, to focus the study on the priority feature. Also we set no thread limitation per group for the same reason.

Alternate high and low priority requests

This experiment aims at evaluating the overhead of our two priority specifications in a simple configuration. In our dedicated multiactive object class, we define five groups from G1 to G5, which have one assigned request each. We declare a linear priority dependency between them: G1 has a higher priority than G2, which has a higher priority than G3, etc. We declare the priorities on one hand using the integer-based specification and on the other hand using the graph-based priority specification. The priority definitions are displayed on Figure 3.15.

```
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G3"}),
    @Set(groupNames = {"G4"}),
    @Set(groupNames = {"G5"})
  })
})
```

```
@DefineIntegerBasedPriorities({
  @Priority(groupNames = {"G1"}, level = 5),
  @Priority(groupNames = {"G2"}, level = 4),
  @Priority(groupNames = {"G3"}, level = 3),
  @Priority(groupNames = {"G4"}, level = 2),
  @Priority(groupNames = {"G5"}, level = 1)
})
```

Figure 3.15: Priority definitions used for the Alternate test case

Both declarations are semantically equivalent, although using different definitions and different structures. In this context, we send in total 1000 requests to the multiactive object. As in the previous test case, we send alternatively a request of highest priority and a request of lowest priority, that is alternatively from group G1 and from group G5. 500 requests are sent per group in total. No request from the other groups is sent, other groups are just meant

to create bigger priority structures. We perform two runs. The first run uses the integer-based definition. The second run uses the graph-based definition. For each run, we average insertion times of requests every 50 consecutive requests sent from the same group. Figure 3.16 shows the results of both priority specifications for the two kinds of requests:

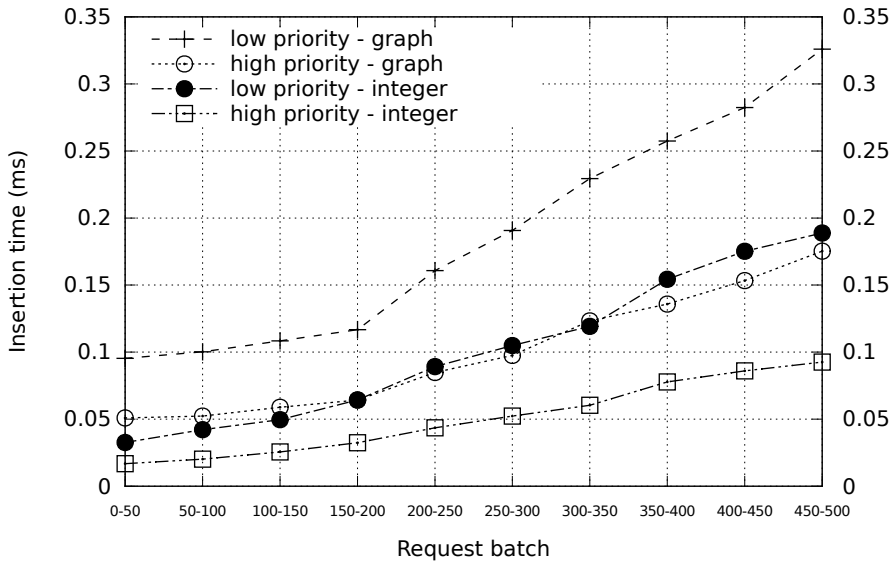


Figure 3.16: Insertion time of alternative low and high priority requests

On this figure, we notice that with both specifications, the insertion time of high priority requests is twice faster than the insertion time of low priority requests, which is an expected behavior. However, there is a gap between the integer-based and the graph-based specification. Indeed, the time to insert a request using the the graph-based specification takes almost twice more time than when using the integer-based specification. This is the case for both high and low priority requests. Using our graph-based specification is thus twice slower in this configuration, because the whole graph must be covered each time the request to be inserted is compared with another request in the ready queue.

To see if this has an impact from the user point of view, we measured the service time of a request. The service time we define is the time to poll the request in the queue and to execute its body in a thread. As we measure quasi-empty request bodies, we measure in fact the minimum time required to execute a request using our framework. We measured an average execution time of 1471303 ns, which is 1.4 ms. In comparison, the insertion of the two hundredth request represents in average only 8% of the service time of a request. In other words, the insertion time is always below 10% of the minimal service time when the ready queue contains less than 200 requests, which is the case for any reasonable application. So even if the graph-based specification is much slower than the integer-based specification in this case, in practice it is still possible to use it without impacting too much the whole execution time.

Sequential requests belonging to a non-naive graph

This experiment illustrates a more realistic test case, where the priority dependencies are not linear, and where insertion time of requests of all groups are evaluated (not only of high and low priority). We define here ten groups, from G1 to G10, each of them having a single belonging request, from g1 to g10. The priority dependencies between them are expanded in Figure 3.17.

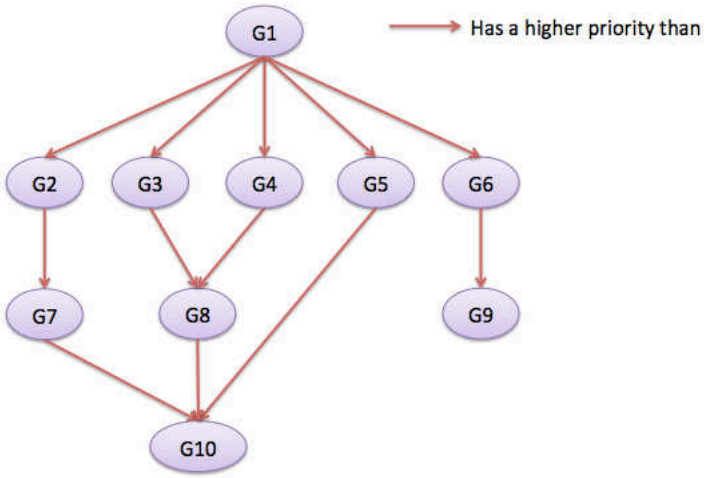


Figure 3.17: Dependency graph used for the Sequential test case

We first declare the corresponding priorities in the code using the graph-based priority specification. We then attempt to declare the same priorities with the integer-based specification. As in this case it is not possible to express the exact same priority relations with the integer-based specification, we declare priorities with integers that have the closest semantic to the dependency graph displayed above. More precisely, this integer-based definition is a *linear extension* of the previous graph-based definition. The two priority declarations that we use are displayed on Figure 3.18.

In this test case, we sequentially send a request of each group in a predefined order. The requests are sent sequentially in the following order (this order has no particular meaning):

- g7, g1, g2, g9, g4, g10, g8, g3, g6, g5

Using a predefined sequence of requests allows us to have deterministic results while experimenting all parts of the graph. We send 500 requests per group, so in total, 5000 requests are sent to the multiactive object. We evaluate here the overhead that we pay in general when using the graph-based priority specification compared to the integer-based priority specification. Unlike the previous experiment, we do not look specifically into the overhead on a given priority, but rather at the global overhead. For that, we average insertion times every 50 consecutive requests sent from the same group. Thus, we have 10 measurements per group, one for each stage of consecutive requests. We finally average the metrics of all groups for

```
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G7"}),
    @Set(groupNames = {"G10"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames ={"G3","G4"}),
    @Set(groupNames = {"G8"}),
    @Set(groupNames = {"G10"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G5"}),
    @Set(groupNames = {"G10"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G6"}),
    @Set(groupNames = {"G9"})
  })
})
```

```
@DefineIntegerBasedPriorities({

  @Priority(groupNames = {"G1"}, level = 4),

  @Priority(groupNames = {"G2"}, level = 3),

  @Priority(groupNames = {"G3"}, level = 3),

  @Priority(groupNames = {"G4"}, level = 3),

  @Priority(groupNames = {"G5"}, level = 3),

  @Priority(groupNames = {"G6"}, level = 3),

  @Priority(groupNames = {"G7"}, level = 2),

  @Priority(groupNames = {"G8"}, level = 2),

  @Priority(groupNames = {"G9"}, level = 2),

  @Priority(groupNames = {"G10"}, level = 1)
})
```

Figure 3.18: Priority definitions used for the Sequential test case

each stage. The results, displayed on Figure 3.19, correspond to the average insertion time of a request of any group.

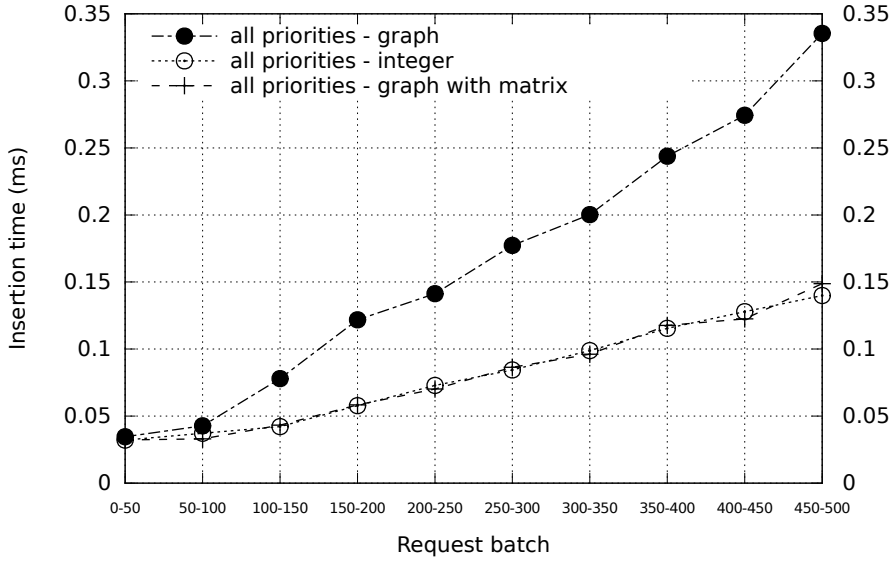


Figure 3.19: Insertion time of sequential requests from a complex priority graph

As the previous usecase, the overhead of the graph-based priority specification on the insertion time is higher than the overhead of the integer-based specification. But in this test case, the difference between the two specifications is bigger than in the previous test case: the

overhead of the graph-based specification is here more than twice higher than the overhead of the integer-based specification. This is explained by the fact that the graph is more complex in this test case. The more complex the graph is (the more nodes and dependencies it has), the bigger the overhead of the graph-based priority specification is, because whenever a request in the ready queue is considered, we must find its corresponding group in the graph, to know if it can be overtaken or not. Even if this overhead is still bearable compared to the service time of a request, we can see that the difference between the two specification grows bigger as the number of requests in the queue increases. So the overhead of the graph-based priority specification could be awkward at some point, even if it is unlikely to happen in real conditions.

To address this problem, we developed an enhanced version of the graph-based priority specification that allows us to reach the performance of the integer-based specification. The idea is to store all the possible combinations of overtakability in a matrix, and to look for a specific entry when needed instead of exploring the graph. The idea is the following. From the dependency graph we build according to the annotations, we build its corresponding graph of *transitive closure*. The transitive closure of a directed graph G is a directed graph G' which has the same nodes and edges as G and where edges are added from any node A that can reach another node B, according to the existing directed edges. In our terminology, we add a directed edge between two nodes A and B if $A \longrightarrow^{+} B$. Figure 3.20 displays the graph of transitive closure of our later test case graph.

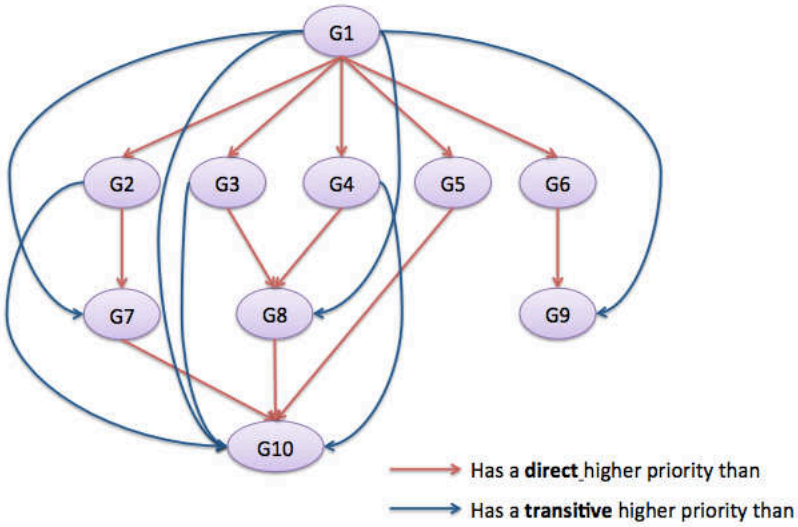
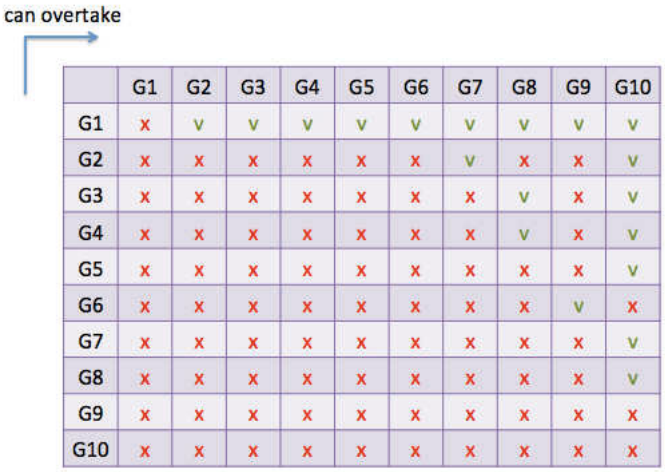


Figure 3.20: Transitive closure of the graph depicted in Figure 3.17

For simplicity, this graph of transitive closure can be seen as a binary matrix. Indeed, considering a matrix M of size NxN where N is the number of nodes, we store a positive value in M[n_1][n_2] if it exists an edge from n_1 to n_2. For example, we can see the corresponding matrix of our test case on Figure 3.21. To speed up the insertion process, we store and access the matrix instead of the graph. When a new request must be inserted in the ready queue, for



	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10
G1	x	v	v	v	v	v	v	v	v	v
G2	x	x	x	x	x	x	v	x	x	v
G3	x	x	x	x	x	x	x	v	x	v
G4	x	x	x	x	x	x	x	v	x	v
G5	x	x	x	x	x	x	x	x	x	v
G6	x	x	x	x	x	x	x	x	v	x
G7	x	x	x	x	x	x	x	x	x	v
G8	x	x	x	x	x	x	x	x	x	v
G9	x	x	x	x	x	x	x	x	x	x
G10	x	x	x	x	x	x	x	x	x	x

Figure 3.21: Overtakability matrix deduced from the transitive closure on Figure 3.20

each request of the ready queue we look the corresponding entry in the matrix, that decides if the incoming request can be placed before the considered request. As the primary graph is built once and used the whole execution time, the matrix is also built once and used the whole execution time, but accessing the matrix is much more efficient than accessing a node in a graph. Indeed, accessing a node in the graph might require to explore the entire graph, which is costly. Consequently, with the matrix, we reach the same performance as with the integer based specification, displayed also on Figure 3.19, because we reduce the overtakability problem to a fundamental operation, as with the integer-based specification.

The only problem is that the matrix is a waste of memory, compared to the graph structure, since most entries have a negative value (whereas the graph only store positive values). There are thus N^2 entries in the matrix, whereas the graph only stores N nodes. This can be a problem if there exists a lot of groups since the graph must be kept in memory the whole execution time. But in practice, the graph has few chances to be highly complex because it would require the multiactive object to have a lot of methods defined. But in a well-designed object-oriented application, objects must have a concise and precise purpose, which is antagonistic with having numerous methods.

Thanks to the matrix representation, we managed to have the same performance as the integer-based specification. In average, its overhead represents 10% of the service time of an empty request. Considering the importance of a priority mechanism and the ease of usage that our graph-based priority specification offers, this overhead is a very small cost. To conclude, the experiment showed that the graph-based priority specification by itself is adapted to a practical usage, and even with strong efficiency requirements, the enhanced version with the reachability matrix can still be used to have the smallest overhead possible. Those experiments show that the graph-based priority specification is efficient in its standard and enhanced versions, in addition to be easy to use and very expressive.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this report, we presented several mechanisms to allow the programmer to control more finely request scheduling of multiactive objects. All mechanisms offer new annotations that we have developed in the Java language. Annotations can be seen as meta programming facilities, and using them is much simpler than coding the intended behavior directly in the business code. This way, scheduling hints are given in a declarative fashion, and then the corresponding code is automatically executed. Related works on high-level scheduling do not provide such an abstract specification, or were fairly complex to use. In most of the cases, they imply to write business code and ask for a large knowledge from the programmer. In our work, we succeeded in providing a high level language to specify scheduling hints, without forcing the programmer to directly manipulate requests in the ready queue.

The first scheduling mechanism that we have introduced gives the possibility to assign different priorities to group of requests, in order to prioritize their execution. For that, we developed two priority specifications. The first priority specification we developed is a classical approach when dealing with priorities: it consists in ordering requests according to the integer order. The programmer uses this order to assign priorities to group of requests. The second priority specification we have developed is more elaborated than the first one, although easier to use. It relies on a dependency graph: when a group has a higher priority than another one, a dependence is created between them. This specification is simpler to use, and more expressive, but requires more processing resources. We developed both specifications in order to evaluate them in practice: we want to select the one that is well balanced between ease of programming and efficiency.

We have also developed other scheduling mechanisms that can be used to control thread allocation in a multiactive object. We have seen that this particular scheduling mechanism was needed to prevent starvation of requests in some cases. The principle of those mechanisms is simple: they consists in defining an upper and lower bound on the number of threads that can be occupied by requests of a given group at a time. A precise partitioning of the threads

using those mechanisms enables a fine-grained control of the request execution. But as they are mainly based on the programmer's consciousness, we chose to override specifications that were not conform to our vision of safety of execution.

Finally, we ran experiments of the priority mechanisms we developed. Firstly, we showed that our graph-based priority mechanism increased significantly the processing of high priority requests. This was a necessary experimentation, simply to show that the results was the ones expected, namely that the priority specification successfully speeded up the execution of high priority requests. The second experimentation showed that our graph-base priority specification introduced a bearable overhead compared to the integer-based priority specification. Indeed, the insertion of a request in the ready queue using the graph represents only a minor part of the execution time of a request. In addition, we have developed an enhanced version of the graph representation that uses the matrix of transitive closure of the initial graph to speed up the insertion process. We showed that this enhancement allows the graph-based specification to reach the same performance as the integer-based specification. To summarize, experiments validated the use of graph-based priorities, because this specification is easier to use, offers a larger expressiveness, and keeps a low overhead (or equal in the enhanced case) compared to classical approaches of priorities.

In conclusion, we enlarged the multiactive object model so that it can benefit from low level concurrency customization and also from high level programming and execution safety. Schematically, we can situate the multiactive object model in between the strict, safe model of active objects, and the very permissive, error prone low level concurrency mechanisms. In this work, we contributed to broaden the scope of multiactive objects towards low level concurrency mechanisms, while trying to keep a safe control over request executions. Therefore, up to now, the multiactive object model is the most complete model to provide both distributed *and* concurrent mechanisms in the same abstraction, while remaining simple and safe.

4.2 Future work

The multiactive object model is still a recent design that needs further work to be finalized. The very next goal is to experiment our scheduling mechanisms in a real case. Indeed, the experiments presented in this work were built to test our scheduling mechanisms at their limits, but we would like now evaluate their impact in a real context. The EventCloud platform [20] is currently developed in the team using multiactive objects. Its goal is to handle notification deliveries in a peer-to-peer manner. We would like to develop a particular usecase using this platform to demonstrate the utility of our scheduling mechanisms.

Also, we would like to explore other ways to manage the threads than the ones we gave. Indeed, our priority mechanism seems to be finalized but our thread management mechanism has not been developed enough to be sure of its optimality, due to lack of time. Although efficient, our thread management mechanism might not be the best to avoid starvation, nor

to ensure a maximum utilization, because it relies on the awareness of the programmer. We will probably have to inspire our work from operating system mechanisms to handle those problems.

In a very long term, the multiactive objects will be the basis of the PhD that I will begin next autumn and that is supervised by Ludovic Henrio [6]. The main goal of this PhD is to design a fault tolerant programming model adapted to globally distributed and locally concurrent programming models. We will use multiactive objects as a concrete implementation of such models. Briefly, the problematic is that fault tolerant protocols for distributed systems have all been designed following the same assumption: messages sent from one entity to another should follow a First In First Out policy. But as multiactive objects introduce other complex scheduling policies, such protocols cannot be used any more to ensure a correct fault tolerance of multiactive objects. Although ad hoc mechanisms exist to handle other policies than FIFO and still get good fault tolerant mechanisms, no general model is provided to handle fault tolerance in a generic manner. Our goal is to converge towards such generic protocols. We will use the scheduling mechanisms developed in this work as a tool to experiment fault tolerant protocols. First, our priority specification will be essential to process in priority requests that are devoted to fault tolerance. Second, we will use our analysis of request scheduling of multiactive objects to deduce a general model for an adaptive fault tolerance of distributed and concurrent systems.

Bibliography

[1] The hats project. `http://www.hats-project.eu`.

[2] Java lesson: Annotations. `http://docs.oracle.com/javase/tutorial/java/annotations/`.

[3] The Play european project. `http://play-project.eu/`.

[4] The proactive java middleware. `http://proactive.inria.fr/`.

[5] The producer/consumer design pattern. `https://en.wikipedia.org/wiki/Producer-consumer`$_{p}$*roblem*.

[6] Programming models and middleware support for distributed and concurrent applications. `http://www-sop.inria.fr/oasis/index.php?page=positionaction=showid=66`.

[7] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1984.

[8] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous sequential processes. *Concurrency and Computation: Practice and Experience*, 2006.

[9] L. Dagum and R Menon. Openmp: an industry standard api for shared-memory programming, 1998.

[10] Edsger Dijkstra. Cooperating sequential processes, 1965.

[11] Tibor Gallai. On directed paths and circuits. *Theory of graphs*, pages 115–118, 1968.

[12] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

[13] Max Haustein and Klaus-Peter Löhr. Jac: declarative java concurrency. *Information and Computation*, 2009.

[14] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded active objects. *Coordination Models and Languages*, 2013.

[15] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 2006.

[16] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs: A core language for abstract behavioral specification. *Formal Methods for Components and Objects*, 2012.

[17] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, February 1980.

[18] R. Greg Lavender and Douglas C. Schmidt. Active object – an object behavioral pattern for concurrent programming, 1995.

[19] Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1883–1888, New York, NY, USA, 2012. ACM.

[20] Laurent Pellegrino, Françoise Baude, and Iyad Alshabani. Towards a Scalable Cloud-based RDF Storage Offering a Pub/Sub Query Service. 2012.

[21] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. *ECOOP 2010 – Object-Oriented Programming*, 2010.

[22] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java system. *Proceedings of the USENIX 1996*, 1996.

Appendix A

Memento of Annotations

Here is an exhaustive list of current annotations for multiactive object, regarding groups, membership, compatibilities, and priorities. Repeatable entities are marked with *. Optional attributes are specified in [ ]. Parameters to be defined by the programmer are marked with P. For each attribute, the expected type of P is mentioned.

```
@DefineGroups({
  @Group*(
    //P: string, name of the group
    name = P,
    //P: boolean, whether requests of this group can be executed at the same time
    selfCompatible = P,
    //P: integer, number of threads occupied at maximum at the same time
    [threadLimit = P],
    //P: integer, number of threads devoted to this group
    [reservedThread = P],
    //P: type, parameter of a request to use to decide on compatibility
    [parameter = P],
    //P: string, boolean method name to decide on selfCompatibility
    [condition = P])
})
```

Figure A.1: Groups - To be used on class entities

```
//P: string, must refer to a group name previously defined.
//Note: the value statement can be omitted (= @MemberOf(P))
@MemberOf(value = P)
```

Figure A.2: Membership - To be used on method entities

```
@DefineRules({
  @Compatible*(
  //P: string[], names of the groups that are compatible together
  //Note: when condition is not used, the value statement can be omitted
  value = P,
  //P: string, boolean method name to decide on compatibility of groups
  [condition = P])
})
```

Figure A.3: Compatibilities - To be used on class entities

```
@DefineGraphBasedPriorities({
  @PriorityOrder*({
    @Set*(
    //P: string[], names of the groups having the same priority relation
    groupNames = P)
  })
})
```

Figure A.4: Priorities (graph) - To be used on class entities

Appendix B

Java 8 Annotations

Java 8 release is planned on early 2014. It will introduce major updates in the annotation mechanism. In particular, when defining an annotation, it will be possible to mark it as *Repeatable* which stipulates that it might be present more than once above an entity, which is not currently allowed. Regarding our annotations, this will greatly lighten the specifications. As an example, Figure B.1 displays the former and future version of a multiactive object definition.

```
@DefineGroups({
  @Group(name="join", ...),
  @Group(name="routing", ...),
  @Group(name="monitoring", ...)
})
@DefineRules({
  @Compatible({"join", "monitoring"),
  @Compatible({"routing", "monitoring")
})
@DefineGraphBasedPriorities({
  @PriorityOrder({
    @Set(groupNames = {"join"}),
    @Set(groupNames = {"monitoring"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"routing"}),
    @Set(groupNames = {"monitoring"})
  })
})
public class Peer {
  @MemberOf("join")
  public JoinResponse join(...) {...}
  @MemberOf("routing")
  public void add(...) {...}
  @MemberOf("routing")
  public Serializable lookup(...) {...}
  @MemberOf("monitoring")
  public void monitor() {...}
}
```

```
@Group(name="join", ...)
@Group(name="routing", ...)
@Group(name="monitoring", ...)

@Compatible({"join", "monitoring"})
@Compatible({"routing", "monitoring"})

@PriorityOrder({
  @Set(groupNames = {"join"}),
  @Set(groupNames = {"monitoring"})
})
@PriorityOrder({
  @Set(groupNames = {"routing"}),
  @Set(groupNames = {"monitoring"})
})
public class Peer {
  @MemberOf("join")
  public JoinResponse join(...) {...}
  @MemberOf("routing")
  public void add(...) {...}
  @MemberOf("routing")
  public Serializable lookup(...) {...}
  @MemberOf("monitoring")
  public void monitor() {...}
}
```

Figure B.1: Java 7 vs Java 8 annotations for multiactive objects